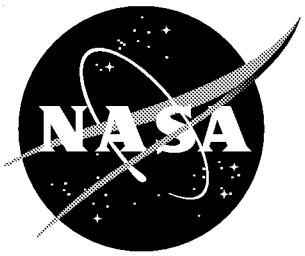
NASA/CR-1999-208991



# A Systematic Methodology for Verifying Superscalar Microprocessors

*Mandayam Srivas*
*SRI International, Menlo Park, CA*

*Ravi Hosabettu & Ganesh Gopalakrishnan*
*University of Utah, Salt Lake City, UT*

February 1999

# The NASA STI Program Office ... in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

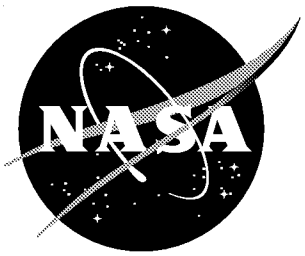
- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7121 Standard Drive
  Hanover, MD 21076-1320

NASA/CR-1999-208991



# A Systematic Methodology for Verifying Superscalar Microprocessors

*Mandayam Srivas*
*SRI International, Menlo Park, CA*

*Ravi Hosabettu & Ganesh Gopalakrishnan*
*University of Utah, Salt Lake City, UT*

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-20334

February 1999

Available from:

NASA Center for AeroSpace Information (CASI)
7121 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

**Abstract**

We present a systematic approach to decompose and incrementally build the proof of correctness of pipelined microprocessors. The central idea is to construct the abstraction function by using *completion functions*, one per unfinished instruction, each of which specifies the effect (on the observables) of completing the instruction. In addition to avoiding the term size and case explosion problem that limits the pure *flushing* approach, our method helps localize errors, and also handles stages with iterative loops. The technique is illustrated on pipelined and superscalar pipelined implementations of a subset of the DLX architecture. It has also been applied to a processor with out-of-order execution.

# Contents

# List of Figures

# Chapter 1

# Introduction

In the past few years, research advance in hardware verification has resulted in the successful verifications of several large and real hardware designs. The verification [SM95] using PVS [ORSvH95] of Rockwell International's AAMP5 and AAMP-FV microprocessors, which was sponsored by NASA's Langley Research Laboratory, was one example of such an effort. While such verification efforts have certainly increased the awareness of the value of formal verification within the hardware design industry, the technology is still far from being successfully and completely transitioned to industry. As the AAMP verification projects demonstrated, the main obstacles to technology transition, especially in microprocessor verification, are the following:

1. Lack of efficient capabilities for symbolic simulation (with uninterpreted functions) of hardware designs and automatic decision procedures for the most commonly used data types in hardware designs, such as bit-vectors.

2. Lack of suitable verification methodologies that are applicable to the kind of challenging architectures, such as superscalar pipelines, out-of-order execution, etc., employed by modern microprocessors.

Support for efficient symbolic simulation is crucial because symbolic simulation is at the core of most of the methods based on commutative diagram correspondence checking used in verifying microprocessor designs at the register-transfer level. Although there exist systems, such as ACL2 [KM96], that support faster symbolic simulation than the current public version of PVS, efficient symbolic simulation alone is not sufficient for scaling up verification to state-of-the-art microprocessors, such as the Pentium

processor. We also need a verification methodology, i.e., appropriate abstraction/refinement techniques and re-usable proof strategies, to setup the overall verification and decompose the complex verification problem into properties that can be automatically verified by symbolic simulation and decision procedures. Under the sponsorship of NASA's Langley Research center, SRI has been working on developing solutions to the above obstacles in scaling up formal verification of microprocessors. This document reports the result of the second task of developing a systematic methodology for verifying microprocessors that employ advanced design features, such as superscalar pipelining, speculative execution, and out-of-order execution to enhance their throughput. Under a separate NASA task, we are enhancing the efficiency and automation capabilities of symbolic simulation in PVS.

Most approaches to mechanical verification of pipelined processors rely on several key techniques. First, given a pipelined implementation and a simpler Instruction Set Architecture (ISA)-level specification, they require a suitable abstraction mapping from an implementation state to a specification state. They use the abstraction function to establish a correspondence between the two machines by means of a commutative diagram. Second, they use symbolic simulation to derive logical expressions corresponding to the two paths in the commutative diagram, which are then tested for equivalence. An automatic way to perform this equivalence testing is to use ground decision procedures for equality with uninterpreted functions such as the ones in PVS. This strategy has been used to verify several processors in PVS [Cyr93, CRSS94, SM95]. Some of the approaches to pipelined processor verification rely on the user providing the definition for the abstraction function. Burch and Dill in [BD94] observed that the effect of flushing the pipeline, for example by pumping a sequence of NOPs, can be used to automatically compute a suitable abstraction function. Burch and Dill used this *flushing approach* along with a validity checker [JDB95, BDL96] (i.e., their version of a decision procedure for uninterpreted functions with equality) to effectively automate the verification of pipelined implementations of several processors.

The pure flushing approach has the drawback of making the size of the generated abstraction function and the number of examined cases impractically large for deep and complex superscalar pipelines. To verify a superscalar example using the flushing approach, Burch [Bur96] decomposed the verification problem into three subproblems and suggested a technique requiring the user to add some extra control inputs to the implementation and set them appropriately to construct the abstraction function. He also had to fine-tune the validity checker used in the experiment, requiring the

user to help it with many manually derived case splits. It is unclear how the decomposition of the proof and the abstraction function used in [Bur96] can be reused for verifying other superscalar examples. Another drawback of the pure flushing approach is that it is hard to use for pipelines with indeterminate latency. Such a situation can arise if the control involves data-dependent loops or if some part of the processor, such as the memory-cache interface, is abstracted away for managing the complexity in verifying a large system.

We propose a systematic methodology to modularize as well as decompose the proof of correctness of microprocessors with complex pipeline architectures. Called the *completion functions* method, our approach relies on the user expressing the abstraction function in terms of a set of completion functions, one per unfinished instruction in the machine. Each completion function specifies the *desired effect* (on the observables) of completing the instruction. Notice that one is not obligated to state *how* such completion would actually be attained, which, indeed, can be very complex, involving details such as squashing, pipeline stalls, and even data-dependent iterative loops. Moreover, we strongly believe that a typical designer would have a very clear understanding of the completion functions, and would not find the task of describing them and constructing the abstraction function onerous. In addition to actually gaining from designers' insights, verification based on the completion functions method has other advantages. It results in a natural decomposition of proofs. Proofs build up in a layered manner where the designer actually debugs the last pipeline stage first through a verification condition, and then uses this verification condition as a rewrite rule in debugging the penultimate stage, and so on. Because of this layering, the proof strategy employed is fairly simple and almost generic in practice. Debugging is far more effective than in other methods because errors can be localized to a stage, eliminating the need to wade through monolithic proofs.

## Related Work

Levitt and Olukotun [LO96] use an "unpipelining" technique for merging successive pipeline stages through a series of behavior preserving transformations. While unpipelining also results in a decomposition of the proofs, their transformation is performed on the implementation, whereas completion functions are defined based on the specification. Their transformations, whihc have been used only for a single issue pipeline, can get complex for superscalar processors and processors with out-of-order execution. Cyrluk's

technique in [Cyr96], which has also been applied to a superscalar processor, tackles the term size and case explosion problem by lazily "inverting the abstraction mapping" to replace big implementation terms with smaller specification terms and using the conditions in the specification terms to guide the proof. Park and Dill have used aggregation functions [PD96], which are conceptually similar to completion functions, for distributed cache coherence protocol verification. In [SH97], Sawada and Hunt used an incremental verification technique to verify a processor with out-of-order execution, which we have reverified with our approach. We describe the differences between the two approaches in section 3.3.

# Chapter 2

# The Completion Functions Approach to Processor Verification

The completion functions approach aims to develop the proof of correctness of pipelined processors in a modular and layered fashion.

## 2.1 Pipelined Microprocessor Correctness Criteria

Figure 2.1(a) shows the correctness criterion (used in [SH97, BD94]) that we aim to establish. Figure 2.1(a) requires that every sequence of `n` implementation transitions that start and end with *flushed* states (i.e., no partially executed instructions) corresponds to a sequence of `m` instructions (i.e., transitions) executed by the specification machine. `I_step` is the implementation transition function and `A_step` is the specification transition function. The `projection` extracts only those implementation state components visible to the specification (i.e., the observables). This criterion is preferred over others where the commute diagram does not necessarily start with a flushed state because it corresponds to the intuition that a real pipelined microprocessor starting at a flushed state, running some program and terminating in a flushed state is emulated by a specification machine whose starting and terminating states are in direct correspondence through projection. This criterion can be proved by induction on `n` once the *commutative diagram* condition shown in Figure 2.1(b) has been proved on a single implementa-

tion machine transition. This inductive proof can be constructed once, as we have demonstrated in the proof files given in [Hos98], for arbitrary machines that satisfy the conditions described in the next paragraph. In the rest of the paper, we concentrate on verifying the commutative diagram condition.

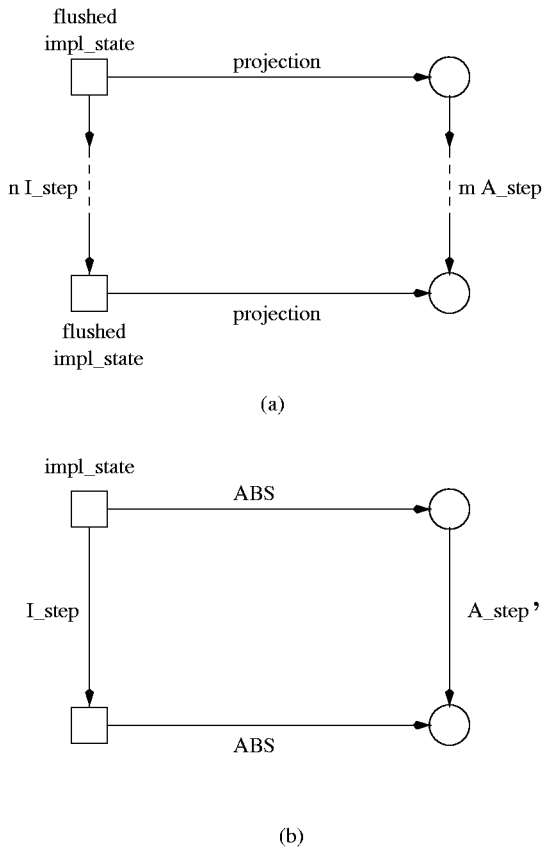



Figure 2.1: Pipelined microprocessor correctness criteria

Intuitively, Figure 2.1(b) states that if the implementation machine starts in an arbitrary reachable state `impl_state` and the specification machine starts in a corresponding specification state (given by an abstraction function `ABS`), then after execution of a transition, their new states correspond. `ABS` must be chosen so that for all flushed states `fs` the *projection condition* `ABS(fs) = projection(fs)` holds. The commutative diagram uses a modified transition function `A_step'`, which denotes zero or more applications of `A_step`, because an implementation transition from an arbitrary

state might correspond to executing in the specification machine zero instructions (e.g., if the implementation machine stalls because of pipeline interlocks) or more than one instruction (e.g., if the implementation machine has multiple pipelines). The number of instructions executed by the specification machine is provided by a user-defined *synchronization* function on implementation states. One of the crucial proof obligations is to show that this function does not always return zero. One also needs to prove that the implementation machine will eventually reach a flushed state if no more instructions are inserted into the machine, to make sure that the correctness criterion in Figure 2.1(a) is not vacuous. In addition, the user may need to discover *invariants* to restrict the set of `impl_state` considered in the proof of Figure 2.1(b) and prove that it is closed under `I_step`.

## 2.2 The Completion Functions Approach

One way of defining `ABS` is to use a part of the implementation definition, modified, if necessary, to construct an explicit *flush* operation [BD94, Bur96]. The completion functions approach is based on using an abstraction function that is behaviorally equivalent to flushing but is *not* derived operationally via flushing.[1] Rather, we construct the abstraction function as a composition (followed by a projection) in terms of a set of *completion functions* that map an implementation state to an implementation state. Each completion function specifies the *desired effect* on the observables of completing a particular unfinished instruction in the machine (assuming those that were fetched ahead of it are completed), leaving all nonobservable state components unchanged. The order in which these functions are composed is determined by the program order of the unfinished instructions. One can use any order that is *consistent*. i.e., that has the same effect, as the program order. The conditions under which each function is composed with the rest, if any, is determined by whether the unfinished instructions ahead of it could disrupt the flow of instructions, for example, by being a taken branch or by raising an exception. Observe that one is not required to state how these conditions are actually realized in the implementation. Any mistakes, either in specifying the completion functions or in constructing the abstraction function, might lead to a false negative verification result, but never a false positive.

Consider a very simple four-stage pipeline with one observable state component `regfile`, which is shown in Figure 2.2. The instructions flow down the pipeline with every cycle in order with no stalls, hazards, and so forth,

[1] Later we discuss a hybrid scheme extension that uses operational flushing.

updating the `regfile` in the last stage. (This is unrealistically simple, but we explain how to handle these artifacts in subsequent sections.) At any time, the pipeline can contain three unfinished instructions, which are held in the three sets of pipeline registers labeled IF/ID, ID/EX, and EX/WB. The completion function corresponding to an unfinished instruction held in a set of pipeline registers (such as ID/EX) defines how the information stored in those registers is combined to complete that instruction. In our example, the completion functions are `C_EX_WB`, `C_ID_EX`, and `C_IF_ID`, respectively. Now the abstraction function, whose effect should be to flush the pipeline, can be expressed as a composition of these completion functions as follows (we omit `projection` here as `regfile` is the only observable state component):

```
ABS(impl_state) = C_IF_ID(C_ID_EX(C_EX_WB(impl_state)))
```

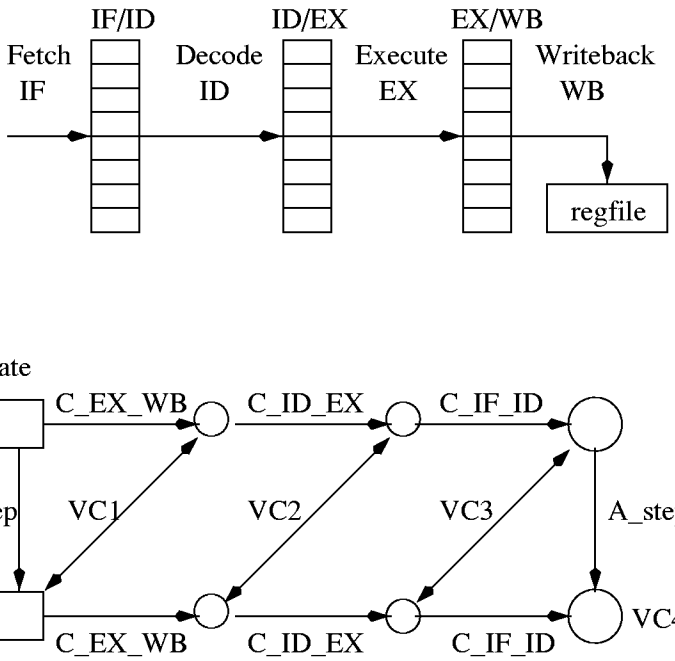



Figure 2.2: A simple four-stage pipeline and decomposition of the proof under completion functions

This definition of the abstraction function leads to a decomposition of the proof of the commutative diagram for `regfile` as shown in Figure 2.2, generating the following series of verification conditions, the last one of which corresponds to the complete commutative diagram:

```
VC1: regfile(I_step(impl_state)) = regfile(C_EX_WB(impl_state))
VC2: regfile(C_EX_WB(I_step(impl_state))) =
            regfile(C_ID_EX(C_EX_WB(impl_state)))
VC3: regfile(C_ID_EX(C_EX_WB(I_step(impl_state)))) =
            regfile(C_IF_ID(C_ID_EX(C_EX_WB(impl_state))))
VC4: regfile(C_IF_ID(C_ID_EX(C_EX_WB(I_step(impl_state))))) =
       regfile(A_step(C_IF_ID(C_ID_EX(C_EX_WB(impl_state)))))
```

The strategy behind the generation of verification conditions uses the fact that `I_step` executes *some part* of each of the instructions already in the pipeline as well as the newly fetched instruction. Each verification condition states the expected effect of `I_step` has in advancing an instruction in the pipeline. This effect can be expressed in terms of the completion functions. For example, VC1 expresses the effect of `I_step` on the instruction in the EX/WB registers: since `regfile` is updated in the last stage, we would expect that after `I_step` is executed, the contents of `regfile` would be the same as after completion of the instruction in the EX/WB registers.

Now consider the instruction in ID/EX. I_step executes it partially as per the logic in stage EX, and then moves the result to the EX/WB registers. `C_EX_WB` can now be used to complete this instruction. This computation must result in the same contents of `regfile` as completion of the instructions held in sets EX/WB and ID/EX of pipeline registers *in that order*. This requirement is captured by VC2. VC3 and VC4 are constructed similarly. Note that our ultimate goal is to prove VC4, with the proofs of VC1 through VC3 acting as "helpers." Each verification condition in the above series can be proved using a *standard strategy* that involves expanding the outermost function on both sides of the equation and using the previously proved verification conditions (if any) as rewrite rules to simplify the expressions, followed by automatic case analysis of the boolean terms appearing in the conditional structure of the simplified expressions. Since we expand only the topmost functions on both sides, and because we use the previously proved verification conditions, the sizes of the expressions produced during the proof and the required case analysis are kept in check.

The completion functions approach also supports *incremental* and *layered* verification. When proving VC1, we are verifying the write-back stage

of the pipeline against its specification `C_EX_WB`. When proving VC2, we are verifying one more stage of the pipeline, and so on. This makes it easier to locate errors. In the flushing approach, if there is a bug in the pipeline, the validity checker would produce a counterexample—a set of formulas potentially involving *all* the implementation variables—that implies the negation of the formula corresponding to the commutative diagram. Such a counterexample cannot isolate the stage in which the bug occurred.

Another advantage of using completion functions is that their definition, unlike that of a flush operation, is not dependent on the latency of the pipeline. Hence, our method is applicable even when the latency of the pipeline is indeterminate. Such a situation can occur when, for example, the pipeline contains data-dependent iterative loops or when the implementation machine has nondeterminism. The proof that the implementation eventually reaches a flushed state can be constructed by defining a measure function that returns the number of cycles the implementation takes to flush and showing that the measure decreases after a transition from a nonflushed state.

A disadvantage of the completion functions approach is that the user must explicitly specify the definitions for these completion functions and then construct an abstraction function. In a later section, we describe a hybrid approach to reduce the manual effort involved in this process.

# Chapter 3

# Application of Our Methodology

In this section, we illustrate the application of our methodology to verify three examples: pipelined and superscalar pipelined implementations of a subset of the DLX processor [HP90] and a processor with out-of-order execution. The DLX example was previously verified in [BD94] using the flushing approach, the superscalar DLX example in [Bur96], and the processor with out-of-order execution in [SH97]. We describe how to specify the completion functions, construct an abstraction function, and handle stalls. We also show the handling of speculative fetching and out-of-order execution, and illustrate the particular decomposition and the proof strategies we used. In Section 3.4, we explain a hybrid approach that reduces the effort in specifying the completion functions in some cases. Our verification is carried out in PVS [ORSvH95]. The detailed implementation, specification, and the proofs for all these examples can be found at [Hos98].

## 3.1 Application to the DLX Processor

The specification of this processor has four state components: the program counter `pc`, the register file `regfile`, the data memory `dmem`, and the instruction memory `imem`. The processor supports six types of instruction: `load`, `store`, unconditional `jump`, conditional `branch`, `alu-immediate` and a three-register `alu` instruction. The ALU is modeled using an uninterpreted function. The memory system and the register file are modeled as stores with `read` and `write` operations.

The implementation uses a five-stage pipeline as shown in Figure 3.1. We

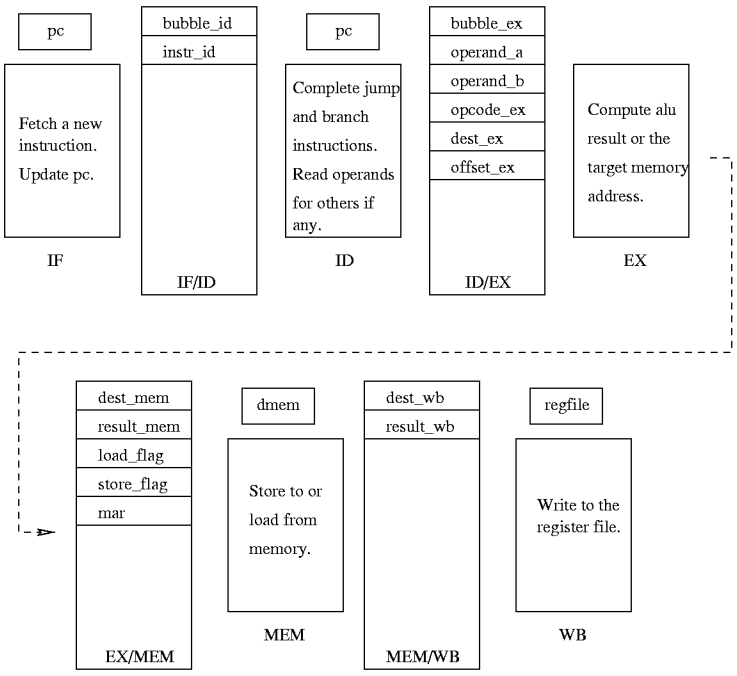



Figure 3.1: Pipelined implementation

organize the fifteen pipeline registers holding information about the partially executed instructions in the design into four sets (shown in columns in Figure 3.1). The intended functionality of each stage is described in words inside the box denoting the stage. The observable components modified in each stage are indicated above the stage (e.g., `pc` is incremented in the IF stage and conditionally modified in the ID stage—and hence is shown twice). The implementation uses a simple "assume not taken" prediction strategy for `jump` and `branch` instructions. Consequently, if a jump or branch is indeed taken (`br_taken` signal is asserted), then the pipeline squashes the subsequent instruction and corrects the `pc`. If the instruction in the IF/ID registers is dependent on a `load` in the ID/EX registers, then that instruction will be stalled for one cycle (`st_issue` signal is asserted); otherwise, the instructions flow down the pipeline with every cycle. No instruction is fetched in the cycle where `stall_input` is asserted. The implementation provides forwarding of data to the instruction decode unit (ID stage) where the operands are read. The details of forwarding are not shown in the figure.

### 3.1.1 Completion Functions and Constructing the Abstraction Function

The processor can have at most four partially executed instructions at any time, one each in the four sets of pipeline registers shown in Figure 3.1. We associate a completion function with each such instruction. We need to identify how a partially executed instruction is stored in a particular set of pipeline registers—once this is done, the completion function for that unfinished instruction can be easily derived from the ISA specification.

Consider the set IF/ID of pipeline registers. The intended functionality of the IF stage is to fetch an instruction (place it in `instr_id`) and increment the `pc`. The `bubble_id` register indicates whether or not the instruction is valid. (It might be invalid, for example, if it is being squashed due to a taken `branch`). So, to complete the execution of this instruction, the completion function should do nothing if the instruction is not valid, otherwise it should update the `pc` with the target address if it is a `jump` or a taken `branch` instruction, update the `dmem` if it is a `store` instruction and update the `regfile` if it is a `load`, `alu-immediate` or `alu` instruction according to the semantics of the instruction. The details of how these operations are done can be obtained from the ISA specification. This function is not obtained by tracing the implementation, instead, the user directly provides the intended effect. Also note that we are not concerned with load interlock or data forwarding while specifying the completion function. We call this function `C_IF_ID`.

1

```
% Complete the unfinished instruction in ID/EX pipeline registers.
Complete_ID_EX(is:state_I):state_I =
   is WITH [ (dmem) :=
               %% Complete the store instruction.
               IF (instr_class(opcode_ex(is)) = store)
                                  AND NOT (bubble_ex(is)) THEN
                  write_dmem(dmem(is),add(operand_a(is),
                                   offset_ex(is)),operand_b(is))
               %% Otherwise leave it unchanged.
               ELSE dmem(is) ENDIF,
             (regfile) :=
               %% Complete the load instruction.
               IF NOT (dest_ex(is)=zero_reg) AND NOT(bubble_ex(is))
                         AND (instr_class(opcode_ex(is)) = load) THEN
                  write_reg(regfile(is),dest_ex(is),read_dmem(dmem(is),
                                    add(operand_a(is),offset_ex(is))))
               %% Complete alu_reg & alu_immed instructions.
               ELSIF NOT (dest_ex(is)=zero_reg) AND NOT (bubble_ex(is))
                      AND ((instr_class(opcode_ex(is)) = alu_reg) OR
                           (instr_class(opcode_ex(is)) = alu_immed)) THEN
                  write_reg(regfile(is),dest_ex(is),
                               alu(alu_op_of(opcode_ex(is)),
                                   operand_a(is),operand_b(is)))
               %% Otherwise leave it unchanged.
               ELSE regfile(is) ENDIF ]

% Complete the unfinished instruction in MEM/WB pipeline registers.
Complete_MEM_WB(is:state_I):state_I =
   is WITH [ (regfile) :=
               %% regfile is the only component updated here.
               IF NOT(dest_wb(is)=zero_reg) THEN
                  write_reg(regfile(is),dest_wb(is),result_wb(is))
               ELSE regfile(is) ENDIF ]
```

Now consider the unfinished instruction in the set ID/EX of pipeline registers. The ID stage completes the execution of `jump` and `branch` instructions, so this instruction would affect only `dmem` and `regfile`. The `bubble_ex` indicates whether or not this instruction is valid, `operand_a` and `operand_b` are the two operands read by the ID stage, `opcode_ex` and `dest_ex` determine the opcode and the destination register of the instruction and `offset_ex` is used to calculate the memory address for `load` and `store` instructions. The completion function should state how these bits of information can be combined to complete the instruction, which again can be gleaned from the specification. We call this function `C_ID_EX`. Similarly, the completion functions for the other two sets of pipeline registers—`C_EX_MEM` and `C_MEM_WB`—are specified. Two of these functions—`C_ID_EX` and

C_MEM_WB—are shown in the PVS code fragment 1.

The completion functions for the unfinished instructions in the initial sets of pipeline registers are very close to the specification and it is very easy to derive them. (For example, C_IF_ID is almost the same as the specification.) However, the completion functions for the unfinished instructions in the later sets of pipeline registers are harder to derive, as the user needs to understand how the information about the instruction is stored in the various pipeline registers, but the functions themselves are usually much more compact. For example, once the designer knows that result_wb holds the result of the instruction in the write-back stage, all C_MEM_WB has to do is to update the register using result_wb. Also the completion functions are independent of how the various stages are implemented and just depend on their functionality.

Since the instructions flow down the pipeline in program order, the abstraction function—which should have the cumulative effect of flushing the pipeline—is defined as a simple composition of these completion functions:

```
ABS(impl_state) =
    projection(C_IF_ID(C_ID_EX(C_EX_MEM(C_MEM_WB(impl_state)))))
```

The synchronization function in this example returns zero if the instruction in IF/ID registers is not issued because of a load interlock, or if no instruction is fetched (because stall_input is asserted), or if the instruction fetched is squashed because of a taken branch; otherwise, it returns one.

2

```
sync(impl_state:state_I): nat =
   IF st_issue(impl_state) OR stall_input(impl_state)
            OR br_taken(impl_state) THEN 0
   ELSE 1 ENDIF
```

### 3.1.2 The Decomposition of the Proof

The decomposition we used for regfile for this example is shown in Figure 3.2. The justification for the first three verification conditions is similar to that given for the example in Section 2.2.

However, in deriving verification conditions for the instruction i in the IF/ID registers it is necessary to consider two separate cases depending on whether or not the instruction could get stalled because of a load interlock. If i is stalled, that is, st_issue is true in impl_state, then I_step will not be advancing, i.e., has no effect on, the instruction i. So, the observables at point 1 in Figure 3.2 should be as though i is not completed—C_IF_ID should be applied in the upper path in the commutative diagram. VC4_r captures this case (condition P1 = st_issue) shown in 3).

VC5_r is for the case when the instruction `i` is issued (so it should be proved under condition P2 = `NOT st_issue`) and is generated similar to the first three verification conditions. Observe that `st_issue` also appears as a disjunct in the synchronization function and hence in `A_step'`. Finally, VC6_r is the verification condition corresponding to the final commutative diagram for `regfile`.

3

```
VC4_r: LEMMA
  FORALL (is:state_I, inp:inputs_type):
  st_issue(is) IMPLIES
    regfile(Complete_ID_EX(Complete_EX_MEM
              (Complete_MEM_WB(I_step(is,inp))))) =
       regfile(Complete_ID_EX(Complete_EX_MEM(Complete_MEM_WB(is))))
```

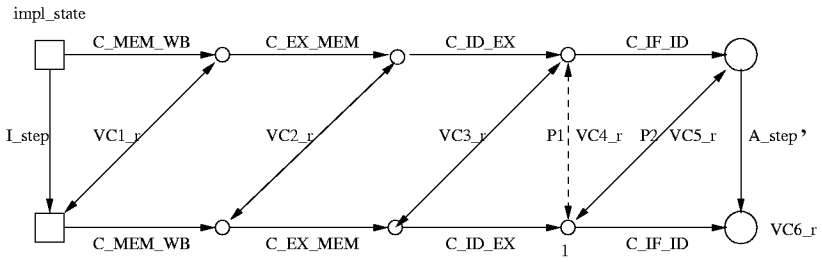



Figure 3.2: The decomposition of the commutative diagram for `regfile`

In general, we generate a separate verification condition for each of the observables, because not every stage modifies every observable. The decomposition used, and hence the VCs generated, for a particular observable depends on the pipeline stages where that observable is updated. For example, the first verification condition VC1_d for `dmem`, shown in 4, states that completing the instruction in the MEM/WB registers has no effect on `dmem` since `dmem` is not updated in the last stage of the pipeline. Other verification conditions are exactly identical to that of `regfile`.

4
```
%% First verification condition for dmem.
VC1_d: LEMMA
  FORALL (impl_state:state_I):
    dmem(C_MEM_WB(impl_state)) = dmem(impl_state)

%% First verification condition for pc.
VC1_p: LEMMA
  FORALL (impl_state:state_I):
    pc(Complete_ID_EX(Complete_EX_MEM(Complete_MEM_WB(impl_state))))
         = pc(impl_state)

%% Second verification condition for pc.
VC2_p: LEMMA
  FORALL (impl_state:state_I):
     NOT st_issue(impl_state) AND NOT br_taken(impl_state) IMPLIES
     pc(Complete_IF_ID(Complete_ID_EX(
         Complete_EX_MEM(Complete_MEM_WB(impl_state))))) =
               pc(impl_state)

%% First verification condition for imem.
VC1_i: LEMMA
  FORALL (impl_state:state_I):
    imem(Complete_IF_ID(Complete_ID_EX(
       Complete_EX_MEM(Complete_MEM_WB(impl_state)))) =
           imem(impl_state)
```

The decomposition for pc has three verification conditions. The last three stages do not modify pc, and this fact is stated by VC1_p (shown in 4). (The three completion functions are combined into one). The second verification condition VC2_p captures the conditions under which the instruction in IF/ID registers does not affect pc and is shown in 4. The third verification condition corresponds to the final commutative diagram for pc.

Finally, the decomposition for imem has two verification conditions. The first one is similar to VC1_p and is shown in 4 and the second one corresponds to the final commutative diagram for imem.

In summary, the decomposition we used has six verification conditions for regfile and dmem, three for pc and two for imem, all systematically generated as explained earlier. Also, this is the particular decomposition that we chose; others are possible. For example, we could have avoided generating and proving, say VC2_r, and proved that goal when it arises within the proof of VC3_r if the prover can handle the term sizes.

### 3.1.3 The Proof Details

The proof is organized into three phases:

- Generating and proving a set of rewrite rules that express certain general properties about the completion functions.
- Proving the verification conditions and other lemmas using the basic rewrite rules.
- Proving other proof obligations mentioned in Chapter 2 including invariants, if needed.

### Rewrite rules about completion functions

These rules express the basic property that the completion functions: should not modify, i.e., map to the same value, the hidden (i.e., nonobservable) variables. For each register in a particular set of pipeline registers, we need a rewrite rule stating that the register is unaffected by the completion functions of the unfinished instructions ahead of it. For example, for `bubble_ex`, the rewrite rule is:
`bubble_ex(C_EX_MEM(C_MEM_WB(impl_state))) = bubble_ex(impl_state)`.
All these rules can be automatically generated (once the completion functions are identified) and automatically proved by rewriting using the definitions of the completion functions. We then define a PVS strategy `setup-rewrite-rules` to make and enter these rules into the prover, and the definitions and the axioms from the implementation and the specification (leaving out a few on which we do case analysis), as rewrite rules.

### Proving the verification conditions and other lemmas

The proof strategy for proving all the verification conditions is similar—use the PVS strategy `setup-rewrite-rules` to install the rewrite rules mentioned earlier, set up the previously proved verification conditions as rewrite rules, `expand` the outermost functions on both sides, use the PVS command `assert` to do the rewrites and simplifications by decision procedures, and then perform case analysis with the PVS strategy `(apply (then* (repeat (lift-if)) (bddsimp) (ground)))`. Minor differences were that some verification conditions (like VC1_d, VC1_p) were proved simply by expanding the definitions of the completion functions, some verification conditions (like VC4_r) needed the outermost function to be expanded on only one side (see 3, expand the first occurrence of `C_ID_EX` and then use VC3_r), and some were slightly more involved (like VC6_r), needing case analysis on the various terms introduced by expanding `A_step'` followed by a similar proof strategy as mentioned above.

The proof above needed a lemma expressing the correctness of the feedback logic. With completion functions, we could state this succinctly as follows:

5
```
% new_operand_a is the value returned by the feedback logic.
% val_a is the value found in the register file.
lemma_new_operand_a: LEMMA
   FORALL (is:state_I):
        NOT stall_issue(is) AND NOT bubble_id(is) IMPLIES
            new_operand_a(is) =
              val_a(Complete_ID_EX(Complete_EX_MEM(Complete_MEM_WB(is))))
```

That is, the value read in the ID stage by the feedback logic (when the instruction in the IF/ID registers is valid and not stalled) is the same as the value read from `regfile` after the three instructions ahead of it are completed. Observe that without completion functions, it would be hard to state the correctness of the feedback logic. Its proof is done by using the strategy `setup-rewrite-rules` to install rewrite rules mentioned earlier, and then setting up the definitions occurring in the lemma as rewrite rules, followed by the PVS command `assert` to do the rewrites and simplifications by decision procedures, followed by `(apply (then* (repeat (lift-if)) (bddsimp) (ground)))` to do the case analysis.

### Other proof obligations

We needed one invariant on the reachable states in this example, and it was discovered during the proof of VC3_r. The proof that the invariant is closed under `I_step` is trivial.

Finally, we prove that the implementation machine eventually goes to a flushed state ("Eventual flush" obligation) if it is stalled sufficiently long enough and then check in that flushed state `fs`, `ABS(fs) = projection(fs)`. For this example, this proof was done by observing that `bubble_id` will be true after two stall transitions (hence no instruction in the IF/ID registers) and that this "no-instruction"-ness propagates down the pipeline with every stall transition. We also need to show that the synchronization function does not always return zero ("No indefinite stutter") and the proof is straightforward.

The table below shows the overall proof organization:

| Proof Obligation | Comments |
|---|---|
| Rewrite rules | Automatically generated. |
| Verification Conditions | 6 each for `regfile` and `dmem`, 3 for `pc` and 2 for `imem`. All systematically generated. |
| Lemma about feedback logic | Uniformly needed in all examples. |
| One invariant<br>"Eventual flush" obligation<br>"No indefinite stutter" obligation | |

## 3.2 Application to Superscalar DLX Processor

The superscalar DLX processor [Bur96] is a dual-issue version of the DLX processor. Both the pipelines have a structure similar to the one shown in Figure 3.1 except that the second pipeline executes only `alu-immediate` and `alu` instructions. In addition, the processor has a one-instruction buffer.

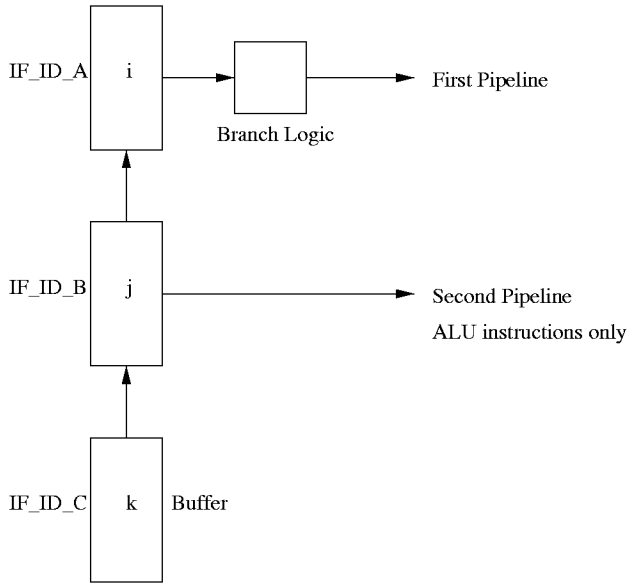



Figure 3.3: The issue logic in the superscalar DLX processor

The issue logic in this processor model, shown in Figure 3.3, is fairly complex—from zero to two instructions can be issued per cycle. Instruction `j` can get stalled because of a load interlock, a dependency on instruction `i`, or because it is neither an `alu-immediate` nor `alu` instruction. If instruction `i` is a taken branch, then instructions `j` and `k` need to be squashed. These factors affect the latency of an instruction waiting to be issued and lead to many scenarios in the proof of the commutative diagram. Once instructions are issued, they flow down the pipeline and complete execution as in the DLX example.

### 3.2.1 Completion Functions and the Abstraction Function

Specifying the completion functions for the various unfinished instructions is similar to the DLX example. This example has nine unfinished instructions, so there are potentially nine completion functions. Since the issued instructions proceed down the pipeline in lockstep, we state the combined effect of completing instructions in the corresponding stages in the two pipelines for the last three stages, and so we have only six completion functions. One main difference is in constructing the abstraction function—we must state how the completion functions of the unfinished instructions (`i`, `j`, and `k`) in the IF/ID registers and the instruction buffer are composed to handle the speculative fetching of instructions. These unfinished instructions could be potential branches since the branch instructions are executed in the first stage of the first pipeline as shown in Figure 3.3. So, while constructing the abstraction function, instruction `j` should be completed only if instruction `i` is not a taken branch. This is as shown in [6], where the completion functions are named `C_i`, `C_j`, and `C_k`). Similarly, instruction `k` should be completed only if instructions `i` and `j` are not taken branches. We used a similar idea in constructing the synchronization function. The specification machine would not execute any new instructions if any of the instructions `i, j, k` mentioned above is a taken branch.

6

```
% Completing the instructions i & j.
% 'rs' should be the composition of the completion functions of
% the instructions ahead of i, in order.

% This predicate tests if instruction i is a taken branch.
branch_taken_pipe_a(rs:real_state) : bool =
   instr_kind_a(rs) = J OR (instr_kind_a(rs) = BEQZ
      AND select(reg(rs),rf1_of(instr_id_a(rs))) = zero)

Complete_IF_ID_AB(rs: impl_state): impl_state =
   IF NOT bubble_id_a(rs) AND branch_taken_pipe_a(rs) THEN
      % Don't complete C_j, if instruction i is a taken branch.
      C_i(rs)
   ELSE
      % If not, complete instruction j.
      C_j(C_i(rs))
   ENDIF
```

It is very easy and natural to express these conditions by using completion functions since we are not concerned with exactly when the branches are taken in the implementation machine. (See, for example, the predicate `branch_taken_pipe_a` above). However, in the pure flushing approach, even the definition of the synchronization function will be much more complicated because it is necessary to cycle the implementation machine for many cycles [Bur96].

### The Differences with the DLX Proof

Because of the complexity of the issue logic in this example, we needed eight additional verification conditions capturing the various scenarios in which the instructions get issued or stalled, or moved around. The proofs of all the verification conditions used similar strategies. The second difference was that the synchronization function had many more cases in this example, and the previously proved verification conditions were used several times during the proof.

## 3.3 Application to Out-of-order Execution

We have applied our methodology to an out-of-order execution processor that was verified by Sawada and Hunt in [SH97]. This processor, shown in Figure 3.4, has three execution units—a multiplier, a load/store unit, and an adder—sharing the write-back stage. The patterned rectangular boxes show the pipeline registers. The structural hazard due to this sharing of the write-back stage is resolved in the issue logic by ensuring there is at most

one instruction attempting to enter the write-back stage in any clock cycle. An `add` instruction takes one cycle in the execution unit, a `load` instruction takes two cycles, and a `mult` instruction takes three cycles. Since there is no reorder buffer, instructions retire immediately after execution. So an `add` instruction, issued immediately after a `mult` instruction, can complete before it. The processor allows such out-of-order execution of an `add` instruction only if its destination register is different from that of the `mult` instruction issued earlier to avoid write-after-write hazards. The processor keeps track of the current instructions executing in the three execution units in the Scheduling Registers block for this purpose.

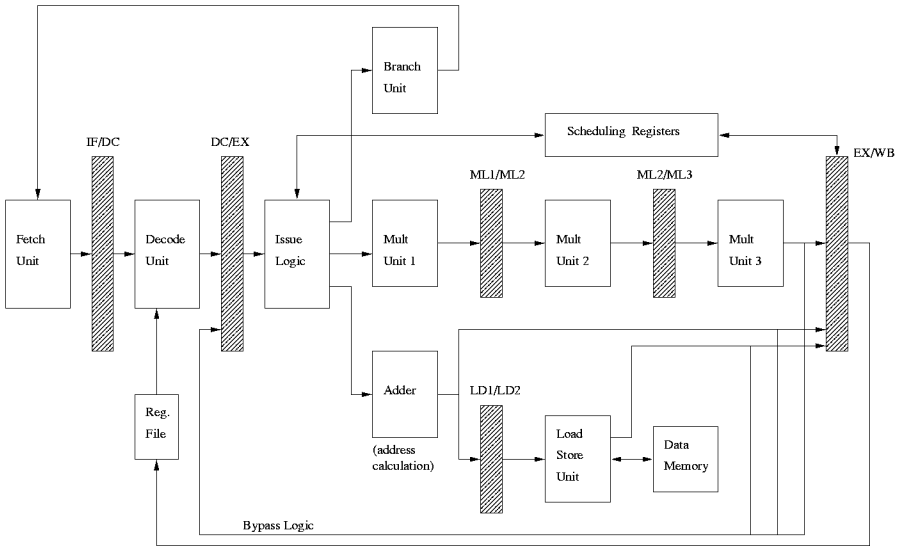



Figure 3.4: The processor with out-of-order execution (example used in [SH97])

### 3.3.1 Constructing the Abstraction Function

The abstraction function for this example is as shown in 7 where the completion functions are named using the same convention as in the previous examples. Note that the completion function for IF/DC and DC/EX stages have been combined into one. The definitions for the completion functions were derived in a fashion similar to the one used in the previous examples. In the previous examples the program order was apparent from the

structure of the pipelines and order in which the pipeline stages were executed. Whereas here, because of the possibility of out-of-order execution, the implementation machine does not have sufficient information to derive the exact program order. For example, the instruction in EX/WB may or may not be ahead of the instruction in ML2/ML3 in the program order. Similarly, the program order of the instructions in LD1/LD2 and ML1/ML2 is unclear. In 7, we have used `Complete_ML2_ML3` after `Complete_EX_WB`, i.e., the stage execution order, in the composition because that order is always guaranteed to consistent with the program order because there cannot be write-after-write hazards. For ML1/ML and LD1/LD2, either order is fine because there can be at most one valid instruction in any given cycle in those registers.

7
```
% Complete_IF_DC_EX completes the instruction in DC/EX
% and then the one in IF/DC
% if the first one is not squashed.
ABS(is:impl_state): abs_state =
       project(Complete_IF_DC_EX(Complete_LD1_LD2(
               Complete_ML1_ML2(Complete_ML2_ML3(Complete_EX_WB(is))))))
```

### 3.3.2 Proof Details

The strategy behind generation of the verification conditions for this processor is based on the observation that four cases arise when considering the instruction about to access the write-back stage—a `mult` instruction in the ML2/ML3 registers, a `load` instruction in the LD1/LD2 registers, an `add` instruction in DC/EX registers that is about to be issued, and none of these three possibilities. That these cases are mutually exclusive follows from the fact that the structural hazard is resolved properly by the issue logic, which is proved as an invariant. We then systematically build the proof of the commutative diagram in the above four cases, formulating and proving the verification conditions as in the earlier examples. The interesting case is the scenario of out-of-order completion when the `add` instruction being issued (from DC/EX registers) bypasses a `mult` instruction issued earlier (and present in ML1/ML2 registers). As mentioned previously, the processor would issue such an `add` instruction only if its destination register is different from that of the `mult` instruction issued earlier. So, though the implementation completes the `add` instruction before the `mult` instruction, one can prove that the net effect is as though `mult` is completed before `add`, that is, the instructions are completed in the order used by the abstraction function. This is captured by the following reordering lemma:

```
lemma_reordering: LEMMA                                                    8
FORALL (is : impl_state): issue_add?(is) IMPLIES
    %% DC/EX has add instruction and ML1/ML2 has a mult instruction.
    reg(Complete_ML1_ML2(Complete_DC_EX(
          Complete_ML2_ML3(Complete_EX_WB(is))))) =
              reg(Complete_DC_EX(Complete_ML1_ML2(
                    Complete_ML2_ML3(Complete_EX_WB(is)))))
```

The other details of the proof, such as, handling the bypass logic and squashing, are similar to the earlier examples.

### 3.3.3 Comparison with the MAETT Approach

In [SH97], Sawada and Hunt construct an intermediate abstraction of the implementation machine by using a table (called MAETT) representing the (infinite) trace of all executed instructions up to the present time. They achieve incrementality by postulating and proving individually a large set of invariant properties about this intermediate representation, from which they derive the final correctness proof. The main difference of our approach is that the incremental nature of the proof in our case arises from the way we construct our abstraction function and the decomposition of the proof of the commutative diagram to which it leads. This decomposition is to a large extent independent of the processor design. Our approach also has the advantage that the amount of information the user needs to specify is significantly less than in their method. For example, we require just a few simple invariants on the reachable states and do not need to construct an explicit intermediate abstraction of the implementation machine.

## 3.4 Hybrid Approach to Reduce the Manual Effort

In some cases, it is possible to *derive* the definitions of some of the completion functions automatically from the implementation to reduce the manual effort. We illustrate this hybrid approach on the DLX example.

The implementation machine is specified in the form of a typical transition function giving the "new" value for each state component as a function of the old values. Since the implementation modifies the `regfile` in the write-back stage, we take `C_MEM_WB` to be `new_regfile`, which is a function of `dest_wb` and `result_wb`. To determine how `C_EX_MEM` updates the register file from `C_MEM_WB`, we perform a step of symbolic simulation of the nonobservables in the definition of `C_MEM_WB`, that is, replace `dest_wb`

and `result_wb` in its definition with their "new-" counterparts. Since the MEM stage updates `dmem`, `C_EX_MEM` will have another component modifying `dmem`, which we simply take as `new_dmem`. Similarly, we derive `C_ID_EX` from `C_EX_MEM` through symbolic simulation. For the IF/ID registers, this procedure gets complicated on two counts: the instruction there that could get stalled because of a load interlock, and the forwarding logic that appears in the ID stage. So, we let the user specify this function directly. We have done a complete proof using these completion functions. The details of the proof are similar. An important difference to note is that the verification with this hybrid approach eliminated the need for the invariant that was needed earlier.

While reducing the manual effort, this way of deriving the completion functions from the implementation has the disadvantage that we are verifying the implementation against itself. This contradicts our view of these as *desired* specifications and negates our goal of incremental verification. To combine the advantages of both, we could use a mixed approach where we use explicitly provided and symbolically generated completion functions in combination. For example, we could derive it for the last stage, specify it for the penultimate stage, then derive it for the stage before that (from the specification for the penultimate stage), and so on.

# Chapter 4

# Conclusions

One of the main obstacles to technology transition in the area of microprocessor verification is the lack of a systematic reusable methodology for refining the verification task into small enough problems that can be discharged automatically. The methodology must work for advanced optimization features that are employed in today's processors. Toward this end, we have developed a systematic approach to modularize and decompose the proof of correctness of pipelined microprocessors with complex controllers to implement design features, such as superscalar pipelining, out-of-order execution, and speculative execution. The overall efficiency and automation of our method depends on the capabilities for symbolic simulation of the underlying verification system. Under a separate NASA task, we are enhancing the efficiency and automation capabilities of symbolic simulation in PVS so that the symbolic simulation speed can be brought to within a few orders of magnitude of conventional simulation speed.

We have shown its generality by applying it to three different processors. The methodology relies on the user expressing the cumulative effect of flushing in terms of a set of completion functions, one per unfinished instruction. This method results in a natural decomposition of the proof based on the individual stages of the pipeline and allows the verification to proceed incrementally, overcoming the term size and case explosion problem of the flushing approach. While this method increases the manual effort on the part of the user, we found that the knowledge required in specifying the completion functions, constructing the abstraction function, and formulating the verification conditions is close to the designer's intuition about how the pipeline works.

One of our future plans is to build a system that uses PVS or a part of

it as a back-end to support the methodology presented. Besides automating parts of the methodology, this system would help the user interactively apply the rest of the process. We would also like to see how our approach can be extended to verify more complex pipeline control that uses reorder buffers or other out-of-order completion techniques. Other plans include testing the efficacy of our approach for verifying pipelines with data dependent iterative loops and asynchronous memory interface.

**Acknowledgments**

We thank John Rushby and David Cyrluk for their feedback on earlier drafts of this report.

# Bibliography


[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David Dill, editor, *Computer-Aided Verification, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford, CA, June 1994. Springer-Verlag.

[BDL96] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In Srivas and Camilleri [SC96], pages 187–201.

[Bur96] J. R. Burch. Techniques for verifying superscalar microprocessors. In *Design Automation Conference, DAC '96*, June 1996.

[CRSS94] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective theorem proving for hardware verification. In Ramayya Kumar and Thomas Kropf, editors, *Theorem Provers in Circuit Design (TPCD '94)*, volume 910 of *Lecture Notes in Computer Science*, pages 203–222, Bad Herrenalb, Germany, September 1994. Springer-Verlag.

[Cyr93] David Cyrluk. Microprocessor verification in PVS: A methodology and simple example. Technical Report SRI-CSL-93-12, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[Cyr96] David Cyrluk. Inverting the abstraction mapping: A methodology for hardware verification. In Srivas and Camilleri [SC96], pages 172–186.

[Hos98] Ravi Hosabettu. PVS specification and proofs of the DLX, superscalar DLX examples and the processor with out-of-order execution, 1998. Available at http://www.csl.sri.com/~ravi/nasa/processor.html.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Mateo, CA, 1990.

[JDB95] R. B. Jones, D. L. Dill, and J. R. Burch. Efficient validity checking for processor verification. In *International Conference on Computer Aided Design, ICCAD '95*, 1995.

[KM96] Matt Kaufmann and J Strother Moore. ACL2: An industrial strength version of Nqthm. In *COMPASS '96 (Proceedings of the Eleventh Annual Conference on Computer Assurance)*, pages 23–34, Gaithersburg, MD, June 1996. IEEE Washington Section.

[LO96] Jeremy Levitt and Kunle Olukotun. A scalable formal verification methodology for pipelined microprocessors. In *Design Automation Conference, DAC '96*, June 1996.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PD96] Seungjoon Park and David L. Dill. Protocol verification by aggregation of distributed actions. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 300–310, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[SC96] Mandayam Srivas and Albert Camilleri, editors. *Formal Methods in Computer-Aided Design (FMCAD '96)*, volume 1166 of *Lecture Notes in Computer Science*, Palo Alto, CA, November 1996. Springer-Verlag.

[SH97] J. Sawada and W. A. Hunt, Jr. Trace table based approach for pipelined microprocessor verification. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 364–375, Haifa, Israel, June 1997. Springer-Verlag.

[SM95] Mandayam Srivas and Steven P. Miller. Formal verification of a commercial microprocessor. Technical Report SRI-CSL-95-4, Computer Science Laboratory, SRI International, Menlo

Park, CA, February 1995. Also available under the title *Formal Verification of an Avionics Microprocessor* as NASA Contractor Report 4682, July, 1995.

# REPORT DOCUMENTATION

Form Approved
OMB No. 0704-0188



| 1. AGENCY USE ONLY (Leave blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | February 1999 | Contractor Report |

**4. TITLE AND SUBTITLE**

A Systematic Methodology for Verifying Superscalar Microprocessors

**5. FUNDING NUMBERS**

C NAS1-20334

W 519-50-11-01

**6. AUTHOR(S)**

Mandayam Srivas
Ravi Hosabettu & Ganesh Gopalakrishnan

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**

SRI International; Menlo Park, CA

University of Utah; Salt Lake City, UT

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23681-2199

**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

NASA/CR-1999-208991

**11. SUPPLEMENTARY NOTES**

Srivas: SRI International, Menlo Park, CA; Hosabettu, Gopalakrishnan: University of Utah, Salt Lake City, UT.
Langley Technical Monitor: Paul S. Miner Final Report

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**

Unclassified-Unlimited
Subject Category: 61 Distribution: Standard
Availability: NASA CASI, (301) 621-0390

**12b. DISTRIBUTION CODE**

**13. ABSTRACT (Maximum 200 words)**

We present a systematic approach to decompose and incrementally build the proof of correctness of pipelined microprocessors. The central idea is to construct the abstraction function by using *completion functions*, one per unfinished instruction, each of which specifies the effect (on the observables) of completing the instruction. In addition to avoiding the term size and case explosion problem that limits the pure *flushing* approach, our method helps localize errors, and also handles stages with interative loops. The technique is illustrated on pipelined and superscalar pipelined implementations of a subset of the DLX architecture. It has also been applied to a processor with out-of-order execution.

**14. SUBJECT TERMS**

Formal Methods, Microprocessor Verification, Superscalar, Theorem Proving

**15. NUMBER OF PAGES**

39

**16. PRICE CODE**

A03

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | |